Modules

Modules may be used to share type definitions, data (parameters and variables), and procedures.

Next slide

The Form of a Module

`module` *module name*

type and data declarations

`contains`

procedures

`end module` *module name*

The procedures in a module are *module procedures*. They may contain internal procedures.

Learn more about module procedures.

Previous slide Next slide

The use Statement

USE *module name*

The use statement may appear in a main program, an extenal, internal, or module procedure, or another module.

Very roughly, the effect of using this statement is the same as placing in the program unit all of the type definitions, declarations, and procedures. However, the values of variables declared are *shared* among all the programs using the module, so changing a value in one program unit may affect the value of the same variable used by another program unit.

Learn more about the use statement.

Previous slide Next slide

```
module m
   integer, parameter :: nr_of_unknowns = 100
   real a (nr_of_unknowns, nr_of_unknowns)
end module m
```

Previous slide Next slide

```
program p
   use m
   implicit none
   . . .
contains

subroutine s
   real, dimension (nr_of_unknowns) :: x
   a = 0
   . . .
end subroutine s

function f(x)  result (f_result)
   real :: x, f_result
   f_result = x + a (1,1)
   . . .
end function f
   . . .
```

Previous slide Next slide

`use only` and Renaming

```
use m, only : nr_of_unknowns
use m, n => nr_of_unknowns
use m, only : n => nr_of_unknowns
```

Previous slide Next slide

The `private` Attribute and Statement

Things with the `private` attribute in a module are not included when the module is `use`d. This means they are known and can be used only inside the module. The `private` statement sets the default attribute to `private` in the module (the default is public).

Learn more about accessibility.

Previous slide Next slide

The `public` Attribute and Statement

In a module, `public` is the default. A common scheme is to make `private` the default with the use of the `private` statement and declare specifically the things to be public with the `public` attribute or statement.

Previous slide Next slide

Compiling Modules

A module can be compiled in a file along with any other program units, but frequently modules are placed in their own files. In this case, they can be compiled separately. It is important to ensure that the correct version of each module is available and compiled during the compilation of any programs that `use` For most compilers, a module being `use`d must be compiled prior to compiling any program that uses the module. Compiling a module typicall produces a .o or .obj file containing the object code for the procedures in the module and files containing interface information for the procedures and information about the types and data in the module. These files typically (but not always) have the suffix ".mod".

Previous slide Next slide

Suppose module1.f90 contains the module `mod_1`. With the compiler from Numercial Algorithms Group (which is typical, but not universal), it can be compiled without creating an executable file with the command:

```
f90 -c module1.f90
```

Then a program in file fff.f90 that uses the module `mod_1` can be compiled with:

```
% f90 module1.o fff.f90
```

You can compile fff.f90 without mentioning module1, but you cannot produce an executable without it.

Previous slide Next slide

Case Study: Numerical Integration

Suppose we now want to integrate a function written as a function subprogram. In this case, we integrate the function

f(x) = sqrt(x) sin (x)

First, put the function in a module.

Previous slide Next slide

```
module function_mod

   implicit none

contains

function f (x)  result (f_result)

   real :: f_result
   real, intent (in) :: x
   f_result = sin (x) * sqrt (x)

end function f

end module function_mod
```

Previous slide Next slide

It is also reasonable to put the integration routine in a module `integral_mod`. Then the program that does the computation is:

```
program integrate

use function_mod
use integral_mod

implicit none

print *, integral (f, a=0.0, b=1.0, n=100)

end program integrate
```

Learn more about modules.

Previous slide Next slide

Generic Procedures

Fortran 77 has generic intrinsic procedures. It is possible to write your own generic procedures in Fortran 90. The easiest way to do it is to put them in a module.

```
module swap_module

   implicit none
   private

   interface swap
      module procedure swap_reals, swap_integers
   end interface
   public swap
```

Learn more about generic procedures.

Previous slide Next slide

```
contains

subroutine swap_reals (a, b)
   real :: a, b, temp
   temp = a; a = b; b = temp
end subroutine swap_reals

subroutine swap_integers (a, b)
   integer :: a, b, temp
   temp = a; a = b; b = temp
end subroutine swap_integers

end module swap_module
```

Previous slide Next slide

```
program test_swap

   use swap_module
   implicit none

   real :: x = 1.1, y = 2.2
   integer :: i = 1, j = 2

   call swap (x, y)
   print *, x, y

   call swap (i, j)
   print *, i, j

end program test_swap
```

Previous slide Next slide

External Generic Procedures

If the subroutines `swap_reals` and `swap_integers` are external subroutines, there must be an interface block in any program that calls them generically as `swap`.

```
subroutine swap_reals (a, b)
   real :: a, b, temp
   temp = a; a = b; b = temp
end subroutine swap_reals

subroutine swap_integers (a, b)
   integer :: a, b, temp
   temp = a; a = b; b = temp
end subroutine swap_integers
```

Previous slide Next slide

```
program test_swap

   implicit none

   interface swap
      subroutine swap_integers (a, b)
      integer a, b
      end subroutine swap_integers
      subroutine swap_reals (a, b)
      real a, b
      end subroutine swap_reals
   end interface
   . . .

end program test_swap
```

Previous slide Next slide

Extending Assignment

```
module int_logical

   implicit none
   private

   interface assignment (=)
      module procedure integer_gets_logical
   end interface
   public assignment (=)

contains
```

Learn more about exting assignment.

Previous slide Next slide

```
subroutine integer_gets_logical (i, l)

   integer, intent (out) :: i
   logical, intent (in) :: l

   i = 0; if (l) i = 1

end subroutine integer_gets_logical

end module int_logical
```

Previous slide Next slide

```
program test_int_logical

   use int_logical
   implicit none
   integer :: i

   i = .false.
   print *, i
   i = (5 < 7) .and. (sin (.3) < 1.0)
   print *, i

end program test_int_logical
```

Previous slide Next slide

Exercise

1. Extend assignment to assign a character string of all digits to an integer. Use the function `int_char` discussed in the ``Character Data'' section.

Previous slide Next slide

Extending Operators

```
module logical_plus

   implicit none
   private

   interface operator (+)
      module procedure log_plus_log
   end interface
   public operator (+)

contains
```

Previous slide Next slide

```
function log_plus_log (x, y)  &
      result (log_plus_log_result)

   logical :: log_plus_log_result
   logical, intent (in) :: x, y

   log_plus_log_result = x .or. y

end function log_plus_log

end module logical_plus
```

Previous slide Next slide

```
program test_logical_plus

   use logical_plus
   implicit none

   print *, .false. + .false.
   print *, .true. + .true.
   print *, (2.2 > 5.5) + (3.3 > 1.1)

end program test_logical_plus
```

Learn more about exting operators.

Previous slide Next slide

Defining New Operators

Defining a new operator is similar to extending an existing one; its name is used in an interface statement and the function, which must have one intent(in) argument is named in a `module procedure` statement.

```
interface operator (.prime.)
   module procedure prime_function
end interface
public operator (.prime.)
```

Previous slide Next slide

This operator could now be used just like any built-in unary operator, as illustrated by the following `if` statement:

```
if (.prime. b .and. b > 100) then
```

The name of an operator must consist of letters only, surrounded by periods, and it must not be the same as any built-in operator (==, `.not.`, `.neqv.`, ...). The precedence of a defined binary operator is always lower than all other operators, and the precedence of a defined unary operator is always higher than all other operators.

Learn more about defining new operators.

Previous slide Next slide

Extending Intrinsic Functions

```
module integer_sqrt

   ! Extended to return the truncated
   ! integer square root of an integer

   implicit none
   private

   interface sqrt
      module procedure sqrt_int
   end interface
   public sqrt

contains
```

Previous slide Next slide

```
function sqrt_int (i) &
      result (sqrt_int_result)

   integer :: sqrt_int_result
   integer, intent (in) :: i

   sqrt_int_result = int (sqrt (real (i) + 0.5))

end function sqrt_int

end module integer_sqrt
```

Previous slide Next slide

A Module for Big Integers

We are interested in adding, multiplying, and dividing very large integers, possibly with hundreds of digits. This can be done by creating a new data type, called `big_integer`, deciding which operations are needed, and writing procedures that will perform the operations on values of this type. All of this will be placed in a module called `big_integers` so that it can be used by many programs.

Previous slide Next slide

To make it easier to conceptualize with simple examples, we will store one decimal digit in each element of a Fortran array of integers.

```
integer, parameter :: nr_of_digits = 100

type, public :: big_integer
   private
   integer, dimension (0 : nr_of_digits) ::  &
         digit
end type big_integer
```

Previous slide Next slide

The array `digit` has 101 elements. `digit(0)` holds the units digit; `digit(1)` holds the tens digit; `digit(2)` holds the hundreds digit; ... . The extra element in the array is used to check for overflow--if any value other than zero gets put into the largest element, that will be considered to exceed the largest `big_integer` value and the program will halt with an error. The `private` statement indicates that we don't want anybody that uses the module to be able to access the *component* `digit` of a variable of type `big_integer`; we will provide all of the operations necessary to compute with such values.

Previous slide Next slide

The first necessary operations assign values to a big integer and print the value of a big integer. This subroutine `print_big` does not have a `use` statement because it will be inside the module `big_integers` and will have access to all the data and procedures in the module.

Previous slide Next slide

```
subroutine print_big (b)

   type (big_integer), intent (in) :: b
   integer :: n

   ! Find first significant digit
   do n = nr_of_digits, 1, -1
      if (b % digit (n) /= 0) exit
   end do

   print "(999i1)", b % digit (n:0:-1)

end subroutine print_big
```

Previous slide Next slide

To be able to assign values to `big_integers` consisting of large integer values, one possibility is to write the integer as a character string consisting of only digits 0-9 (we are not allowing negative numbers). If `c` contains a character other than one of the digits, the program halts with an error.

Previous slide Next slide

```
subroutine big_gets_char (b, c)

   type (big_integer), intent (out) :: b
   character (len = *), intent (in) :: c
   integer :: n, i

   if (len (c) > nr_of_digits) then
      b = huge (b)
      return
   end if
```

Previous slide Next slide

```
   b % digit = 0
   n = 0
   do i = len (c), 1, -1
      b % digit (n) =  &
         index ("0123456789", c (i:i)) - 1
      if (b % digit (n) == -1) then
         b = huge (b)
         return
      end if
      n = n + 1
   end do

end subroutine big_gets_char
```

Previous slide Next slide

Putting the Procedures in a Module

A module using what we have created so far follows. We also need to extend the intrinsic function `huge` to apply to big integers. This is done later.

```
module big_integers

   implicit none
   private
   integer, parameter :: nr_of_digits = 100

   type, public :: big_integer
      private
      integer, dimension (0 : nr_of_digits) ::  &
            digit
   end type big_integer

   interface huge
      module procedure huge_big
   end interface
   public huge
```

Previous slide Next slide

```
contains

subroutine print_big (b)

   type (big_integer), intent (in) :: b
   integer :: n

   ! Find first significant digit
   do n = nr_of_digits, 1, -1
      if (b % digit (n) /= 0) exit
   end do

   print "(999i1)", b % digit (n:0:-1)

end subroutine print_big
```

Previous slide Next slide

```
subroutine big_gets_char (b, c)

   type (big_integer), intent (out) :: b
   character (len = *), intent (in) :: c
   integer :: n, i

   if (len (c) > nr_of_digits) then
      b = huge (b)
      return
   end if
```

Previous slide Next slide

```
   b % digit = 0
   n = 0
   do i = len (c), 1, -1
      b % digit (n) =  &
      index ("0123456789", c (i:i)) - 1
      if (b % digit (n) == -1) then
         b = huge (b)
         return
      end if
      n = n + 1
   end do

end subroutine big_gets_char

function huge (b) result (huge_result)
   type (big_integer), intent (in) :: b
   . . .
end function huge

end module big_integers
```

Previous slide Next slide

With the module available, we can write a simple program to test out the assignment and printing routines for big integers.

```
program test_big_1

   use big_integers
   implicit none
   type (big_integer) :: b1
```

Previous slide Next slide

```
   call big_gets_char  &
      (b1, "712345678090987654321")
   call print_big (b1)

   call big_gets_char (b1, "")
   call print_big (b1)

   call big_gets_char  &
      (b1, "123456789+987654321")
   call print_big (b1)

end program test_big_1

run test_big_1

71234567890987654321

0

999999999999999999 . . .
```

Previous slide Next slide

Assigning Big Integers

It is possible to use the assignment statement to do the conversion from character to big integer. Here is what the interface block must look like in this case.

```
interface assignment (=)
   module procedure big_gets_char
end interface
public assignment (=)
```

Now any user of the module can use the assignment statement instead of calling a subroutine, which makes the program a lot easier to understand.

Previous slide Next slide

```
program test_big_2

   use big_integers
   implicit none
   type (big_integer) :: b1

   b1 = "712345678909876543 21"
   call print_big (b1)
   print *
   b1 = ""
   call print_big (b1)
   print *
   b1 = "123456789+987654321"
   call print_big (b1)

end program test_big_2
```

Previous slide Next slide

With conversion from character strings to big integers available using the assignment statement, there is no need to have the subroutine `big_gets_char` available. This can be done by putting the `private` statement in the module to make the default accessiblity `private` and put the following `public` statement in the module.

```
public assignment (=)
```

The effect of this `private` statement is different from that of the `private` statement that occurs within the definition of the type `big_integer`. This one makes everything except those with the `public` attribute inaccessible outside the module, whereas the `private` statement in the type statement makes only the components of the type inaccessible outside the module. Both the type definition and the procedure are accessible inside the module.

Previous slide Next slide

Extending Intrinsic Functions to Big Integers

The intrinsic function `huge` can be extended so that when given a big integer as argument, it returns the largest possible big integer.

Previous slide Next slide

```
function huge_big (b) result (huge_big_result)

   type (big_integer) :: huge_big_result
   type (big_integer), intent (in) :: b

   huge_big_result % digit  &
         (0 : nr_of_digits - 1) = 9
   huge_big_result % digit (nr_of_digits) = 0

end function huge_big
```

Previous slide Next slide

This function is tested by the program `test_big_5`.

```
program test_big_5

   use big_integers
   implicit none
   type (big_integer) :: b

   print *, huge (b)

end program test_big_5
```

which produces the output

```
99999999999999999999999999 . . .
```

There is not room enough on one line to show all 100 9s in the answer.

Previous slide Next slide

Adding Big Integers

Adding big integers can be done with a function that does just what we do with pencil and paper, adding two digits at a time and keeping track of any carry, starting with the rightmost digits. The function `big_plus_big` does this.

```
function big_plus_big (x, y)  &
      result (big_plus_big_result)

   type (big_integer) :: big_plus_big_result
   type (big_integer), intent (in) :: x, y
   integer :: carry, temp_sum, n
```

Previous slide Next slide

```
   carry = 0
   do n = 0, nr_of_digits
      temp_sum = &
         x % digit (n) + y % digit (n) + carry
      big_plus_big_result % digit (n) =  &
            modulo (temp_sum, 10)
      carry = temp_sum / 10
   end do

   if (big_plus_big_result % digit (nr_of_digits)  &
         /= 0 .or. carry /= 0) then
      big_plus_big_result = huge (x)
   end if

end function big_plus_big
```

Previous slide Next slide

In mathematics, the symbols + and - are used to add and subtract integers. It is possible to extend the generic properties of the operations already built into Fortran.

```
interface operator (+)
   module procedure big_plus_big
end interface
public operator (+)
```

Previous slide Next slide

The use of the plus operator to add two big integers is tested by the program `test_big_3`.

```
program test_big_3

   use big_integers
   implicit none
   type (big_integer) :: b1, b2

   b1 = "12345678909876543210"
   b2 = "98765432101234567890"
   call print_big (b1 + b2)

end program test_big_3

run test_big_3

111111111011111111110
```

Previous slide Next slide

It is not possible to use the expression `b + i` in a program where `b` is a big integer and `i` is an ordinary integer. To do that, we must write another function and add its name to the list of functions in the interface block for the plus operator. Similarly, it would be necessary to write a third function to handle the case `i + b`. Even if that is not done, the number 999 could be added to `b` using the statements

```
temp_big_integer = "999"
   b = b + temp_big_integer
```

Previous slide Next slide

Precedence of Extended Operators

Similar interface blocks and functions can be written to make the other operations utilize symbols, such as - and *. The precedence of the extended operators when used to compute with big integers is the same as when they are used to add ordinary integers. This holds true for all built-in operators that are extended. This is illustrated by the following program that tests the extended multiplication operator (the function is not shown). By looking at the last digit of the answer, it is possible to see that the multiplication is done before the addition.

Previous slide Next slide

```
program test_big_4

   use big_integers
   implicit none
   type (big_integer) :: a, b, c

   a = "1"
   b = "99999999999999999"
   c = "99999999999999999"
   call print_big (a + b * c)

end program test_big_4

run test_big_4

9999999999999999980000000000000000002
```

Previous slide Next slide

Raising a Big Integer to an Integer Power

Exponentiation has both an iterative definition and a recursive definition. They are

$x^n = x\ x\ x\ x$ ... n times

and

$x^0 = 1$
$x^n = x\ x^{n-1}$ for $n > 1$

Previous slide Next slide

Since Fortran has an exponentiation operator (**) for real numbers, it is not necessary to write a procedure to do that. However, it may be necessary to write an exponentiation procedure for a new data type, such as our big integers. We suppose that the multiply operator (*) has been extended to form the product of two big integers. The task is to write a procedure for the module that will raise a big integer to a power that is an ordinary integer. This time, the simple iterative procedure is presented first.

Previous slide Next slide

```
function big_power_int (b, i)  &
      result (big_power_int_result)

   type (big_integer) :: big_power_int_result
   type (big_integer), intent (in) :: b
   integer, intent (in) :: i
   integer :: n

   big_power_int_result = "1"
   do n = 1, i
      big_power_int_result =  &
            big_power_int_result * b
   end do

end function big_power_int
```

Previous slide Next slide

A recursive definition that leads to a more efficient algorithm is:

$x^0 = 1$
$x^n = (x^{n/2})^2$ for n even, $n > 0$
$x^n = x(x^{n/2})^2$ for n odd, $n > 0$

where / indicates integer division.

Previous slide Next slide

```
recursive function big_power_int (b, i)  &
      result (big_power_int_result)

   type (big_integer) :: big_power_int_result
   type (big_integer), intent (in) :: b
   integer, intent (in) :: i
   type (big_integer) :: temp_big

   if (i <= 0) then
      big_power_int_result = "1"
   else
```

Previous slide Next slide

```
      temp_big = big_power_int (b, i / 2)
      if (modulo (i, 2) == 0) then
         big_power_int_result =  &
               temp_big * temp_big
      else
         big_power_int_result =  &
               temp_big * temp_big * b
      end if
   end if

end function big_power_int
```

Learn more about recursion.

Previous slide Next slide

Exercises

1. Extend the the equality operator (==) and the less than (<) operator to compare two big integers. Test these operators.
2. Extend the equality operator (==) to compare a big integer with a character string consisting of digits. Hint: use extended assignment to assign the character string to a temporary big integer, then use the extended equality operator from Exercise 1 to do the comparison.
3. Extend the multiplication operator (*) to two big integers.
4. Extend the subtraction operator (-) so that it performs ``positive'' subtraction. If the difference is negative, the result should be 0.

Previous slide Next slide

5. The representation of big integers used in this section is very inefficient because only one decimal digit is stored in each Fortran integer array element. It is possible to store a number as large as possible, but not so large that when two are multiplied, there is no overflow. This largest value can be determined portably on any system with the statements:

```
integer, private, parameter :: &
      d = (range (0) - 1) / 2,  &
      base = 10 ** d

! Base of number system is 10 ** d,
! so that each "digit" is 0 to 10**d - 1
```

Modify the module `big_integers` to use this representation.

6. Compute 100!

Previous slide Next slide

7. Project: Write a module to do computation with rational numbers. The rational numbers should be represented as a structure with two integers, the numerator and the denominator. Provide assignment, some input/output, and some of the usual arithmetic operators. 8. Modify the module in the previous exercise to use `big_integer`s for the numerator and denominator. 9. Project: Write a module to manipulate big decimal numbers such as

```
284473058301393757503 02.37429125612092391 23
```

using the `big_integer` module as a model.

Previous slide